Costing JIT Traces

John Magnus Morton, Patrick Maier, and Philip Trinder

University of Glasgow
j.morton.2@research.gla.ac.uk, {Patrick.Maier, Phil.Trinder}@glasgow.ac.uk

Abstract. Tracing JIT compilation generates units of compilation that are easy to analyse and are known to execute frequently. The AJITPar project aims to investigate whether the information in JIT traces can be used to make better scheduling decisions or perform code transformations to adapt the code for a specific parallel architecture. To achieve this goal, a cost model must be developed to estimate the execution time of an individual trace.
This paper presents the design and implementation of a system for extracting JIT trace information from the Pycket JIT compiler. We define three increasingly parametric cost models for Pycket traces. We perform a search of the cost model parameter space using genetic algorithms to identify the best weightings for those parameters. We test the accuracy of these cost models for predicting the cost of individual traces on a set of loop-based micro-benchmarks. We also compare the accuracy of the cost models for predicting whole program execution time over the Pycket benchmark suite. Our results show that the weighted cost model using the weightings found from the genetic algorithm search has the best accuracy.

1 Introduction

Modern hardware is increasingly multicore, and increasingly, software is required to exhibit decent parallel performance in order to match the hardware's potential. Writing performant parallel code is non-trivial for a fixed architecture, yet it is much harder if the target architecture is not known in advance, or if the code is meant to be portable across a range of architectures. Existing approaches to address this problem of *performance portability*, e.g. OpenCL [19], offer a device abstraction yet retain a rather low-level programming model designed specifically for numerical data-parallel problems.

In contrast, problems of a more symbolic nature, e.g. combinatorial searches, computational algebra, are not well supported. These problems often do exhibit large degrees of parallelism but the parallelism is *irregular*: The number and size of parallel tasks is unpredictable, and parallel tasks are often created dynamically and at high rates.

The *Adaptive Just-in-Time Parallelism* (AJITPar) project [1] investigates a novel approach to deliver *portable parallel performance* for programs with irregular parallelism across a range of architectures by combining declarative task parallelism with dynamic scheduling and dynamic program transformation.

Specifically, AJITPar proposes to adapt task granularity to suit the architecture by transforming tasks at runtime, varying the amount of parallelism. To facilitate dynamic transformations, AJITPar will leverage the dynamic features of the Racket language and its recent trace-based JIT compiler, Pycket [9, 6].

Dynamic task scheduling and dynamic task transformation both require the prediction of task runtimes. In this paper, we investigate how to construct simple cost models to predict task runtimes in AJITPar. These cost models essentially attempt to predict the execution time of *traces*, i.e. linear paths through the control flow that the compiler has identified as being executed often. Due to the very restricted control flow of traces, we hypothesize that even very simple cost models can yield reasonably accurate runtime predictions.

The main contributions in this paper are as follows. We have designed and implemented a system for extracting JIT trace information from Pycket JIT compiler (Section 3). We have defined 3 cost models for JIT traces (Section 3.4), ranging from very simple to parametric, and we have used an automated search to tune the cost model parameters on a Pycket benchmark suite. We have also classified the typical instruction mixes of 32000 Pycket traces generated by 26 programs from the benchmark suite (Section 4). We have used this information to produce a number of varying length synthetic benchmarks, in order to compare and validate the cost models. We also compare accuracy for predicting whole program execution time by analysing 43 programs from the Pycket benchmark suite (Section 6).

We outline our plans for ongoing and future work (Section 7).

2 Background

2.1 AJITPar

The Adaptive Just-in-time Parallelism Project (AJITPar) [1] investigates whether *performance portability* for *irregular parallelism* can be achieved by dynamically transforming the program for a particular architecture, using information from JIT trace analysis. The idea is that a programmer only has to write a parallel program once, and then AJITPar's tool will automatically apply code transformations at runtime to enable parallelism on any platform the program is running on from a specific range of platforms. AJITPar aims to investigate to what degree the use of JIT compilation and dynamic scheduling helps achieve this.

AJITPar uses a high-level sub-language for expressing parallelism semi-explicitly, in that decisions about where parallelism should occur are specified by the programmer, but task scheduling decisions are left to the runtime system. To enable performance portability for irregular parallelism, JIT compilation will be leveraged. The lack of complex control-flow in a JIT trace makes it particularly amenable to static analysis; cost analysis can be performed on a trace to estimate the execution time of it - this can be used to produce a picture of the granularity of a task and the irregularity of the parallelism. This "static" analysis can actually be performed at runtime because of the relatively small size

and simplicity of traces; information is also available at runtime which is not available statically.

A JIT compiler also allows dynamic transformations to be performed, since the essence of a JIT is dynamic recompilation. AJITPar aims to identify different transformations which can be used to expose a suitable degree of parallelism for the architecture and degree of irregularity.

AJITPar also proposes a scheduling system for parallelism. This will dynamically parallelise tasks based on timing information from the JIT.

The work described in this paper aims to identify a system for calculating relative costs of traces, which will be used to determine the scheduling of parallel tasks based on their relative costs, and the selection of appropriate transformations to optimise for the parallel work available in the task.

2.2 Tracing JIT

Interpreter-based language implementations, where a program is executed upon a virtual machine rather than on a real processor are often used for a variety of reasons - including ease of use, dynamic behaviour and program portability, but are often known for their poor performance compared to statically compiled languages such as C or FORTRAN.

JIT compilation is a technology that allows interpreted languages to significantly increase their performance, by dynamically compiling well-used parts of the program to machine code. This enables interpreters or virtual machine languages to approach performance levels reached by statically compiled programs without sacrificing portability. Dynamic compilation also allows optimisations to be performed which might not be available statically.

JIT compilation does not compile the entire program as it is executed, rather it compiles small parts of the program which are executed frequently (these parts are described as *hot*). The most common compilation units are functions (or methods) and traces [5]. A trace consists of a series of instructions which make up a path through the body of loop. A complete trace contains no control-flow except at the points where execution could leave the trace; these points are known as *guards*. The main benefit of traces compared to functions as a unit of compilation is that it can form the entire body of a loop spanning multiple functions, rather than just the body of a single function.

2.3 RPython Tool-chain

PyPy[28] is an alternative implementation of the Python programming language[18], notable for having Python as its implementation language. PyPy is implemented using a subset of the Python language known as RPython and the tool-chain is intended to be used as a general compiler tool-chain. Smalltalk[8] and Ruby[24] are examples of languages implemented on the RPython tool-chain. PyPy has a trace-base JIT, and the RPython tool-chain allows for the JIT to be easily added to a new interpreter implementation

Pycket[9] is an implementation of the Racket language built on PyPy's tool-chain. Racket is a derivative of the Scheme Lisp derivative [27] with a number of extra features. Pycket uses the Racket front-end to compile a subset of Racket to a JavaScript Object Notation (JSON)[22] representation of the abstract syntax tree (AST) and uses an interpreter built with the RPython tool-chain to interpret the AST.

JITs built with RPython are notable in that they are *meta-tracing*[7]. Rather than trace an application level loop, the JIT traces the actual interpreter loop itself. The interpreter will annotate instructions where an application loop begins and ends in order for appropriate optimisations to be carried out. The purpose of this is so that compiler writers do not need to write a new JIT for every new language that targets RPython/PyPy, they just provide annotations.

2.4 Cost Analysis

Static analysis techniques can be used to estimate the run-time cost of executing a program. This is particularly important when the cost of instrumenting running code is unfeasibly high or hard real-time and embedded systems are being dealt with.

To estimate the cost of a program or piece of program without running it, some sort metric is required. One of the most well known code metric is *cyclomatic complexity*, first described by McCabe [23], which uses control flow analysis to estimate the complexity of a program based on the number of different paths through it. This measure was not intended for use in estimating the performance of the code, but is meant as a measure of the complexity of the code in engineering and maintainability terms.

In embedded and real-time software systems, the most important performance metric is the *worst case execution time*, as strict timing information is necessary. Various tools[29] have been built to statically estimate or measure this; an example is aiT from Ferdinand and Heckmann [17] which uses a combination of control flow analysis and lower level tools, such as cache and pipelining analysis. Cache and pipelining analysis attempts to predict the caching and processor pipelining behaviour of a program and is performed in aiT using abstract interpretation.

High level cost analysis can also be performed on the syntactic structure of the source code of a program, as described by Brandolese et al. [12] who use a mathematical function of C syntactic constructs to estimate execution time.

Cost analysis and resource analysis in general are the subject of continuing research presented at the biennial Foundational and Practical Aspects of Resource Analysis (FOPARA)[16].

Brady and Hammond [11] describe a system for estimating execution time by encoding information on the size of function arguments in a dependent type system.

Many of the currently available approaches for statically estimating execution time are based around control flow analysis[29][17]; since execution traces from

a JIT compiler do not contain any control flow, these types of analysis are redundant and a much more specialised approach is required.

2.5 Code Transformation

Program transformations are central to optimising compilers. GHC, for instance, aggressively optimises Haskell code by equational rewriting[20][21].

Transformations can also be used for optimising for parallel performance. Algorithmic skeletons[14] - high level parallel abstractions or design patterns - can tuned by code transformations to best exploit the structure of input data or to optimise for a particular hardware architecture. Examples of this include the PMLS compiler[26], which tunes parallel ML code by transforming skeletons based on offline profiling data, and the Paraphrase Project's refactorings[13] and their PARTE tool for refactoring parallel Erlang programs[10].

3 Trace Cost Analysis

3.1 Pycket Trace Structure

A JIT *trace* consists of a series of instructions recorded by the interpreter, as discussed previously.

Crucial to understanding the operation of the JIT is the concept of *hotness*. A loop is considered *hot* if the number of jumps back to the start of the loop is higher than a given threshold, indicating that the loop might be executed frequently and is thus worth compiling.

Other important concepts include *guards*, assertions which cause execution to leave the trace when they fail; *bridges*, which are traces starting at a guard that fails often enough; and *trace graphs*, representing sets of traces. The nodes of a trace graph are entry points (of loops or bridges), labels, guards, and jump instructions. The edges of a trace graph are directed and indicate control flow. Note that control flow can diverge only at guards and merge only at labels or entry points. A *trace fragment* is a part of a trace starting at a LABEL and ending at a JUMP, at a GUARD with a bridge attached, or at another LABEL, with no LABEL in between.

The listing in Figure 1 shows a Racket program incrementing an accumulator in a doubly nested loop, executing the outer loop 10^5 times and the inner loop 10^5 times for each iteration of the outer loop, thus counting to 10^{10}.

Figure 1 also shows the trace graph produced by Pycket. The inner loop (which becomes hot first) corresponds to the path from l2 to j1, and the outer loop corresponds to the bridge. Note that the JIT unrolls loops once to optimise loop invariant code, producing the path from l1 to l2.

The trace graph is a convenient representation to read off the trace fragments. In this example, there the following four fragments:

l1 to l2, l2 to g2, l2 to j1, and l3 to j2. Note that trace fragments can overlap: for instance, l2 to j1 overlaps l2 to g2.

```
(define numb1 100000)
(define numb2 100000)

(define (inner iter acc)
  (if (> iter numb2)
      acc
      (inner (+ iter 1) (+ acc 1))))

(define (outer iter acc)
  (if (> iter numb1)
      acc
      (outer (+ iter 1) (inner 0 acc))))

(outer 0 0)
```

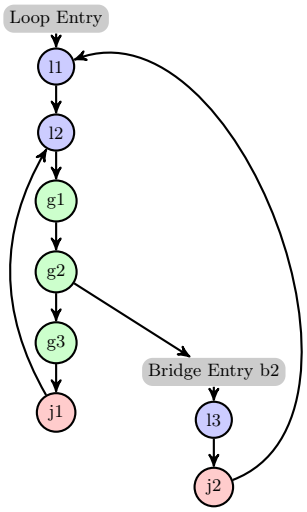

Fig. 1: Doubly nested loop in Racket and corresponding Pycket trace graph.

Figure 2 shows a sample trace fragment, l2 to j1, corresponding to the inner loop. Besides debug instructions, the fragment consists of 3 arithmetic-logical instructions and 3 guards (only the second of which fails often enough to have a bridge attached).

The label at the start brings into scope 3 variables: the loop counter i7, the accumulator i13, and a pointer p1 (which plays no role in this fragment). The jump at the end transfers control back to the start and also copies the updated loop counter and accumulator i15 and i16 to i7 and i13, respectively.

3.2 Runtime Access to Traces and Counters

The RPython tool chain provides language developers with a rich set of APIs to interact with their generic JIT engine. Among these APIs are a number of callbacks that can intercept intermediate representations of a trace, either straight after recording, or after optimisation. We use the latter callback to gain access to the optimised trace and run our trace cost analysis.

Additionally, RPython (in debug mode) can instrument traces with counters, counting every time control reaches an entry point or label. RPython provides means to inspect the values of these counters at runtime. AJITPar will use this feature in the future to derive estimates of the cost of whole loop nests from the cost and frequency of their constituent trace fragments. For now, we dump the counters as the program terminates and use this information to evaluate the accuracy of trace cost analysis (Section 6).

The JIT counts the number of times a label is reached but we are more interested in counting the execution of trace fragments. Fortunately, we can work out the trace fragment execution count due to the fact that there is a one-to-one

```
label(i7, i13, p1, descr=TargetToken(4321534144))
debug_merge_point(0, 0, '(let ([if_0 (> iter numb2)]) ...)')
guard_not_invalidated(descr=<Guard0x10196a1e0>) [i13, i7, p1]
debug_merge_point(0, 0, '(> iter numb2)')
i14 = int_gt(i7, 100000)
guard_false(i14, descr=<Guard0x10196a170>) [i13, i7, p1]
debug_merge_point(0, 0, '(if if_0 acc ...)')
debug_merge_point(0, 0, '(let ([AppRand0_0 ...] ...) ...)')
debug_merge_point(0, 0, '(+ iter 1)')
i15 = int_add(i7, 1)
debug_merge_point(0, 0, '(+ acc 1)')
i16 = int_add_ovf(i13, 1)
guard_no_overflow(descr=<Guard0x10196a100>) [i16, i15, i13, i7, p1]
debug_merge_point(0, 0, '(inner AppRand0_0 AppRand1_0)')
debug_merge_point(0, 0, '(let ([if_0 (> iter numb2)]) ...)')
jump(i15, i16, p1, descr=TargetToken(4321534144))
```

Fig. 2: Trace fragment l2 to j1.

correspondence between guards and their bridges. Essentially, the frequency of a fragment ℓ to g is the frequency of the bridge attached to guard g. The frequency of a fragment starting at ℓ and not ending in a guard is the frequency of label ℓ minus the minus the frequency of all shorter, overlapped trace fragments starting at ℓ. Table 1 and Table 2 demonstrate this on the trace fragments of the nested loop example. The first two columns show the JIT counters, the remaining three columns show the frequency of the four trace fragments, and how they are derived from the counters. Note that not all counters reach the values one would expect from the loop bounds. This is because counting only starts once code has been compiled; iterations in warm-up phase of the JIT compiler are lost. The hotness threshold is currently 131 for loops.

JIT counter	JIT count
n_{l1}	100,001
n_{l2}	10,000,098,957
n_{b2}	99,801
n_{l3}	99,800

Table 1: JIT counters and counts for program in Figure 1.

3.3 Instruction Classes

When discussing the cost models, it is useful to classify the RPython JIT instructions into different sets. We begin with the set of all instructions *all*. Initially, it

fragment	frequency expression	frequency
l1 to l2	n_{l1}	100,001
l2 to g2	n_{b2}	99,801
l2 to j1	$n_{l2} - n_{b2}$	9,999,999,156
l3 to j2	n_{l3}	99,800

Table 2: JIT counters and trace fragment frequencies for program in Figure 1.

was decided to sub-divide *all* into two subsets: the set debug instructions *debug* and all other instructions; this is based on the idea that debug operations are removed by optimisations and do not count towards runtime execution costs.

It was further theorised that some instructions will be more costly than others. The set of all non-debug instructions was further subdivided into high-cost instructions *high* and low-cost instructions *low*, based on their expected relative performance.

Class	Example Instructions
debug	debug_merge_point
numeric	int_add_ovf
guards	guard_true
alloc	new_with_vtable
array	arraylen_gc
object	getfield_gc

Table 3: Instruction classes

Further classification of the instructions can be made based on the conceptual grouping of them and makes no assumptions of their performance characteristics. The classes are object read and write instructions *object*, guards *guards*, numerical instructions *numeric*, memory allocation instructions *alloc* and array instructions *array*. These classes are described in Table 3. Jump instructions are ignored, since there is only ever one in a trace. External calls are excluded as two foreign function calls could do radically different things.

3.4 Cost Models

A cost model — a model or tool for estimating the real world resource use for a program or part of a program — for a JIT trace can be viewed as an execution of the trace with an alternative cost semantics. Different models can be produced by assigning varying costs to each instruction.

In our system, a cost model is implemented by as a method which takes a trace listing and returns a dimensionless number, the *predicted trace cost*. In this paper, we describe four cost models.

All of the cost models can be considered a function of a trace and an equation is provided for each. In each equation, γ is the cost function, n is the number of

instructions on the trace, and the names for the instruction classes are the same as those in Section 3.3.

Null Cost Model (CM_0) The simplest possible cost model for a trace simply returns the same cost for each trace, regardless of the instructions contained. The purpose of this model is to serve as baseline to compare others to; it is not intended to be used in a real system. Using this model to calculate the cost for whole programs can be considered roughly equivalent to using control-flow analysis for estimating the execution time of a program.

$$\gamma = 1 \tag{1}$$

Counting Cost Model (CM_C) The first model simply counts the number of instructions in the trace, excluding debug operations. The idea is that the cost of traces is proportional to their length. This cost model is described formally by Equation (2).

$$\gamma = \sum_{i=0}^{n} \begin{cases} 0, & \text{if } x_i \in debug \\ 1, & \text{otherwise} \end{cases} \tag{2}$$

Weighted Cost Model (CM_W) Certain types of instructions are likely to have a greater effect on the cost of a trace than others, since, for example, memory accesses are several orders of magnitude slower than register accesses. This suggests that the simple counting cost model is not likely to be the most accurate estimate of the real cost of a trace.

A new cost model is created by applying a weighting factor to each of the instruction classes described in section 3.3. An abstract definition of this model is shown in equation 3.

$$\gamma = \sum_{i=0}^{n} \begin{cases} 0, & \text{if } x_i \in debug \\ a, & \text{if } x_i \in array \\ b, & \text{if } x_i \in numeric \\ c, & \text{if } x_i \in alloc \\ d, & \text{if } x_i \in guard \\ e, & \text{if } x_i \in object \end{cases} \tag{3}$$

Whole Program Cost If γ is the cost of a single trace, then the cost of a whole program consisting of a number of traces is as described in equation 4.

$$\Gamma(P) = \sum_{i=1}^{m} n_i \gamma(x_i) \tag{4}$$

where n_i is the ith trace counter and x_i is the ith trace.

4 Pycket Benchmark Suite Analysis

Given that the results in Section 6.1 indicate that traces are dominated by "high-cost" instructions, it is prudent to check that this is true in the general case. To achieve this we again look at the cross-implementation benchmarks from Pycket-bench. We exclude programs which include operations which relate to the foreign function interface, since external calls have unpredictable run times. This results in all string benchmarks being excluded

4.1 Whole Suite Analysis

A histogram of JIT operations, taken from traces generated by all the cross-implementation benchmarks and shown in Figure 3, shows that overall these traces are also dominated by "high-cost" instructions.

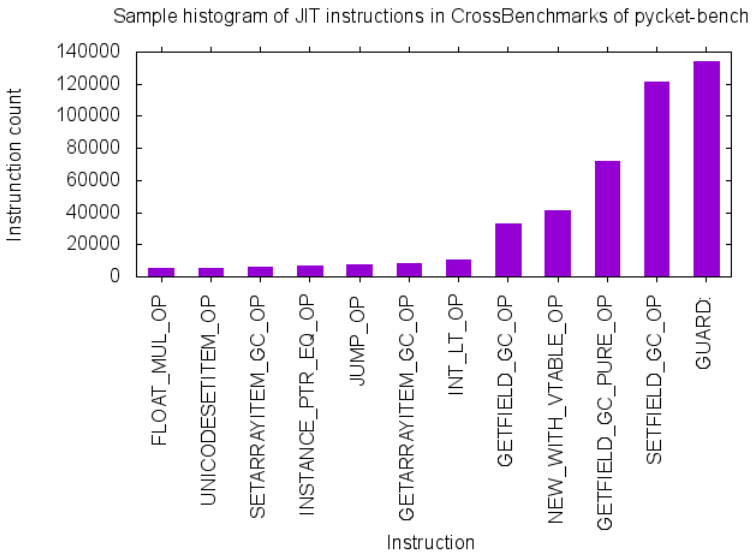

Fig. 3: Most common instructions in cross-implementation Pycket benchmarks

4.2 Program-level Analysis

Individual programs in the Pycket benchmarks suite show quite varying instruction distributions compared to that shown in Figure 3, though they are still dominated by guards and object operations. Using k-means analysis, these programs can be divided into two clusters: numeric and non-numeric. The numeric programs still have a significant proportion of object operations. In Table 4 Cluster 1 contains nearly all numerical benchmarks.

Cluster 1	Cluster 2
ack, array1,fib, fibc, pnpoly, sum, sumloop, trav2, fibfp, sumfp,	boyer, cpstak, ctak, dderiv, deriv, destruc, diviter, divrec, lattice, nboyer, perm9, primes, puzzle, sboyer, tak, takl

Table 4: Clusters for whole benchmarks

4.3 Trace-level Analysis

Looking at the 32013 individual trace fragments in the Pycket benchmark suite, a lot more variation is seen compared to the variation between the whole program histograms. *k-means* clustering shows 3 distinct clusters, the centroids of which are shown in Table 5.

Traces in cluster 1 outnumber both 2 and 3 combined and are again dominated by object instructions and guards. From the centroid of cluster 2 we can see that the proportion of allocation instructions is much higher; this could correspond to the "cleaning up" portion of a trace where previously unboxed primitives are boxed again or possibly `cons` functions calls. Cluster 3 contains a higher proportion of array and numerical instructions.

Cluster	count	object (%)	array (%)	numeric (%)	alloc (%)	guards (&)	jumps (%)
1	17946	38	0.15	0.69	5.6	51	4.0
2	9934	59	0.12	0.51	19	16	4.8
3	4133	22	4.8	11	1.6	54	6.8

Table 5: Trace fragment centroids

5 Cost Model Search

To use the abstract weighted cost model CM_W (section 3.4), it is necessary to find values for each of the five weight parameters in equation 3. Rather than simply guess at appropriate values, we can systematically search the parameter space for an optimal solution. To do this a set of benchmarks are required, along with a search approach and a means of checking the accuracy of the cost model.

5.1 Performance Benchmarks

Using the cross-implementation benchmark suite from pycket-bench (section 4), with the addition of the Racket *Programming Languages Benchmark Game*[2]

benchmarks, the execution times and trace logs for each benchmark are recorded. The execution times are the average of 10 runs.

The platform is an Ubuntu 15.04 system with an Intel Core i5-3570 quad-core 3.40 GHz processor and 16GB of RAM. The Pycket version is revision 5d97bc3f of the `trace-analysis` branch of our custom fork[3], built with Racket version 6.1 and revision 72b01aec157 of PyPy. A slightly modified version of pycket-bench is used.

5.2 Model Accuracy

By applying a instance of a cost model to the trace output from the benchmark runs, the execution time for each benchmark can be plotted against the cost for that benchmark calculated using equation 4.The accuracy of the cost model is calculated by applying linear regression to the plot to obtain a linear best fit. The value of r^2, or the coefficient of determination[15], is used as an estimate of model accuracy; the higher the value the better the fit, and therefore the more accurate the cost model. The linear regression calculation is implemented using the SciPy library to enable automation.

5.3 Exhaustive Search

An exhaustive search of a part of the weight parameter space can be carried out by systematically varying the weights in equation 3. Representing the weights as a vector $\langle a, b, c, d, e \rangle$, the search covers all integral vectors between $\langle 0, 0, 0, 0, 0 \rangle$ and $\langle 10, 10, 10, 10, 10 \rangle$. On termination, the search returns the weight vector for the most accurate cost model (i.e. the model with the highest r^2 coefficient) in the given parameter space.

5.4 Genetic Algorithm Search

Unfortunately, the search space of the exhaustive search grows very quickly with the size of the bounds on the weight parameter space. While a bound of 10 is still feasible, exhaustively searching a paramter space to a bound of 100 is no longer possible. Fortunately metaheuristic search methods allow large search spaces to be covered relatively quickly.

Genetic Algorithms(GA) [25] are a set of meta-heuristics applied to search problems which attempt to mimic natural selection. Rather than exhaustively search the problem space, genetic algorithms attempt to evolve an optimal solution from an initial population. Genetic algorithms use a *fitness function* to evaluate the quality of a solution. The search process consists of a number of generations, in which the entire population is evaluated according to the fitness function and the fittest surviving to the next generation or being selected to reproduce and generate children for the next generation. Reproduction involves selecting any number of solutions from the population and combining them to produce a new solution which contains aspects of its "parents". Random mutation is added to increase the diversity of the population. The search can be run

for a fixed number of generations, until a sufficiently optimal solution is found, or until the population converges.

Genetic Algorithms are chosen as our metaheuristic as the coefficient of determiniation r^2 of linear regression is a useful fitness function, and the vector components map well to the idea of a "chromosome". Details of the search procedure are as follows.

- The *population* is a set of 40 weight parameter vectors $\langle a, b, c, d, e \rangle$. The first generation is completely random; subsequent generations are produced by selection, crossover and mutation, as described below.
- The *fitness function* is simply the r^2 value from the linear regression of the benchmark execution times against the benchmark costs (according to the cost model beign evaluated).
- Each new generation contains the fittest vector from the previous generation. Other vectors in each generation are created by
  1. *selecting* two parent vectors from the previous generation by "tournament selection" (where the fittest of two randomly chosen vectors survives to become a parent),
  2. producing a child vector by *crossing over* the parent vectors componentwise at random, and
  3. randomly *mutating* components of the child vector at a rate of 10%.
- The search terminates at 30,000 generations, returning the weight vector for the most accurate model found so far.

Subsampling Many of the benchmarks in the benchmark suite are intended to test specific Scheme language features or JIT performance, and some benchmarks perform markedly different from the majority when analysed with the null and counting cost models CM_0 and CM_C. This raises the possibility that the benchmark suite contains outliers that will weaken the the linear regression of any possible weighted cost model. We use random sub-sampling whereby 8 randomly selected benchmarks are removed from each search, in order to account for the possibility of outliers in the benchmark suite. The best cost model reported is the best model found for *any* of 125 tested benchmark samples.

5.5 Search Results

Exhaustive Search The cost model found by exhaustive search is displayed in equation 5.

$$\gamma = \sum_{i=0}^{n} \begin{cases} 0, & \text{if } x_i \in array \cup guard \cup debug \cup object \\ 1, & \text{if } x_i \in numeric \\ 10, & \text{if } x_i \in alloc \end{cases} \tag{5}$$

Genetic Algorithm Search The cost model found by Genetic Algorithm search and subsampling is described in equation 6.

$$\gamma = \sum_{i=0}^{n} \begin{cases} 0, & \text{if } x_i \in debug \\ 34, & \text{if } x_i \in array \\ 590, & \text{if } x_i \in numeric \\ 9937, & \text{if } x_i \in alloc \\ 14, & \text{if } x_i \in guard \\ 211, & \text{if } x_i \in object \end{cases} \tag{6}$$

$$\gamma = \sum_{i=0}^{n} \begin{cases} 0, & \text{if } x_i \in debug \\ 2.43, & \text{if } x_i \in array \\ 42.1, & \text{if } x_i \in numeric \\ 709.8, & \text{if } x_i \in alloc \\ 1.00, & \text{if } x_i \in guard \\ 15.1, & \text{if } x_i \in object \end{cases} \tag{7}$$

The benchmark sample excluded the benchmarks ack, divrec, fib, fibfp, heapsort, lattice, tak, and trav2.

The normalised version of this cost model, where the smallest non-zero weight is one, is shown in equation 7. This is similar to the cost model found with using exhaustive search; the ratio between the allocation and numeric weighting is 16.84 in equation 6 and 10 in equation 5.

The cost model in Equation 7 suggests that allocation instructions are the greatest contributor to program execution time followed by numeric instructions. The relatively high weighting of the numerical instructions in this model is interesting, as numerical computation is expected to take significantly less time than the reads and writes seen in object operations; however, the fact that numeric types are required to be boxed and unboxed, resulting in allocations and object reads and writes could account for this weighting.

6 Cost Model Evaluation

Our main hypothesis is that there is a linear relationship between modeled trace cost and execution time, forming the following equation:

$$t(x) = k\gamma(x) \tag{8}$$

where x is a trace, k is some constant, $t(x)$ is the execution time for the trace and $\gamma(x)$ is the cost function for the trace.

If Equation (8) holds, we expect that the total program runtime T will be related linearly to Γ, as defined in equation 4:

$$T = k\Gamma(P) \tag{9}$$

To evaluate a cost model it must be applied to real programs and evaluated against timing information. By timing a piece of code which corresponds exactly to a known trace fragment, we can calculate the average execution time for that trace and plot it against the modeled cost for that trace; plotting time/cost data for enough traces should provide the k of Equation (8) as the gradient of a fitted line. All experiments are run using the same platform as in section 5.1.

6.1 Generated Benchmarks

It is difficult to reliably obtain timing data for single traces using real programs. Even very simple programs result in multiple traces and working out how much of a program's execution time belongs to which trace is very difficult. Fortunately, we can sythesise benchmarks which contain only a single trace. To reliably obtain benchmarks with progressively longer traces, we leverage Racket's macro system to dynamically generate three synthetic single-trace benchmark programs. The benchmarks consist of a sequence of operations in the body of a for loop; the length of the body and the chosen instructions can be changed by changing values in the macro. Care has to taken to chose operations which would not be optimised away by the JIT compiler. For the first benchmark, the operations of the loop body update a vector with the result of a multiplication of random numbers; all random numbers are chosen by the macro ahead of time to avoid calls to a random number generator at runtime. The second benchmark extends the first by adding integer comparisons to the loop body, and the third benchmarks adds list cons operations on top of that.

The maximum trace length produced by the generated program was checked against the maximum trace length found in the cross-implementation benchmarks in the pycket-bench benchmark suite [4] to ensure that the traces covered the length range of "natural" traces. The results are summarised in Table 6.

Benchmark	k (CM_C)	k (CM_W)
Vector Update	9672×10^{-6}	2.85×10^{-6}
Vector Update,Comparison	14760×10^{-6}	4.457×10^{-6}
Vector Update, Comparison, Lists	10780×10^{-6}	3.557×10^{-6}

Table 6: k for different benchmarks

6.2 Whole Program Benchmarks

To attempt to validate Equation (9) it is necessary to evaluate the cost models against more complex benchmarks. To this end, the programs which had no foreign-function calls from the pycket-bench cross-platform and shootout benchmark suites were used - these are the same benchmarks from section 5, but with the outliers identified by random subsampling (Section 5.5) removed. The total cost for each benchmark, computed according to Equation (4), is plotted against the total execution time for that benchmark.

Plots Figures 4, 5 and 6 show the plot and linear regression for cost models CM_0, CM_C and CM_W respectively. The cost models have increasing better determination coefficients (r^2): 0.34, 0.34 and 0.55. The points for benchmarks excluded by subsampling are shown for reference.

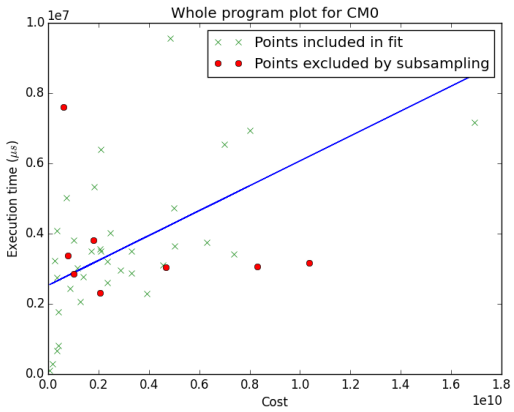

Fig. 4: Execution time against program cost for CM_0

A plot of the residuals from the least-squares regression for each cost model is presented in Figure 7. This illustrates how each cost model fits individual benchmarks. The residuals excluded by subsampling are also present shown here,faded and marked with stars; note that some of these, such as lattice and trav2 are quite small residual values, suggesting that they are not actually outliers.

6.3 Discussion

The coefficients of determination (r^2) values for the whole program benchmark plots show that CM_W clearly fits the data better than CM_C and CM_0, having a value of 0.552 compared to 0.343 for CM_C and 0.342 for CM_0. CM_C and CM_0 have similar values for r^2 suggesting that simply counting instructions is not very useful when looking at entire programs. The values from the generated benchmarks show k values which show roughly an order of magnitude difference from those found in the whole program evaluation. This is most likely due to tracing overhead in the whole program benchmark as different benchmarks have different numbers of traces and bridges, each of which could take varying amounts of time to become hot, whereas the generated benchmarks all contain a single trace, with the same number of executions each time. Other reasons could include the limited types of operations in the generated benchmarks compared to the whole program benchmarks.

7 Discussion and Ongoing work

We have designed and implemented a system for extracting JIT trace information from the Pycket JIT compiler (Section 3). We have defined 3 cost models for JIT traces, CM_0, CM_C and CM_w ranging from the extremely simple CM_0, via a relatively simple costing model CM_C to the weighted CM_W. We have

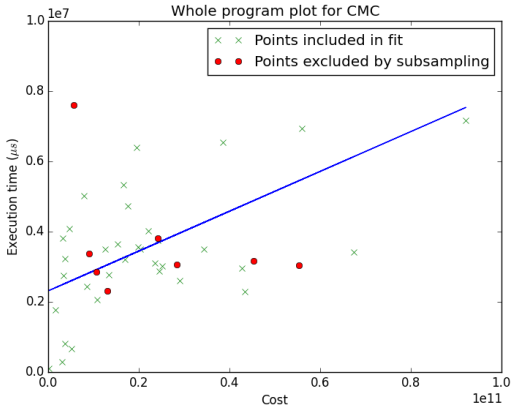

Fig. 5: Execution time (μs) against program cost for CM_C

applied a genetic algorithm to the 43 programs in the Pycket benchmark suite to determine appropriate weight parameters for CM_W (Section 5). We have classified the typical instruction mixes of 32000 Pycket traces generated by 26 programs from the benchmark suite (Section 4). We use the classification to produce a number of varying length synthetic benchmarks, in order to compare and validate the cost models.

We compare the accuracy of the cost models for predicting whole program execution time by analysing the Pycket benchmark programs and find that the weighted CM_W produces a markedly better fit than the simpler CM_C and CM_0 models. Moreover CMW shows a linear fit when applied to single traces similar k value found between traces from different classes of program (Section 6).

We argue that the weighted CM_W cost model that we have developed is appropriate for Pycket JIT traces, and speculate that similar techniques can be used to identify an analogous models for the traces produced by the JIT implementations of other languages, e.g. Java, Javascript etc. In future work in the AJITPar project we will use the execution cost predictions provided by CM_W to direct the transformation of parallel Pycket programs to adapt to specific parallel hardware.

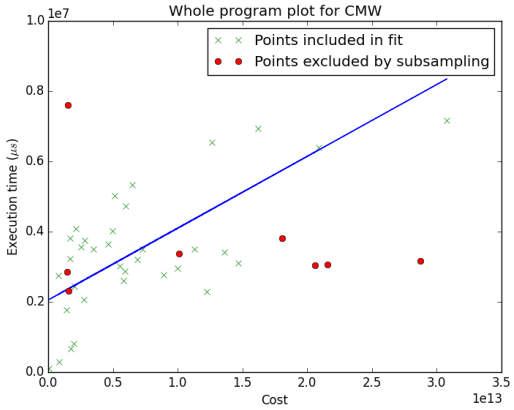

Fig. 6: Execution time (μs) against program cost for CM_W

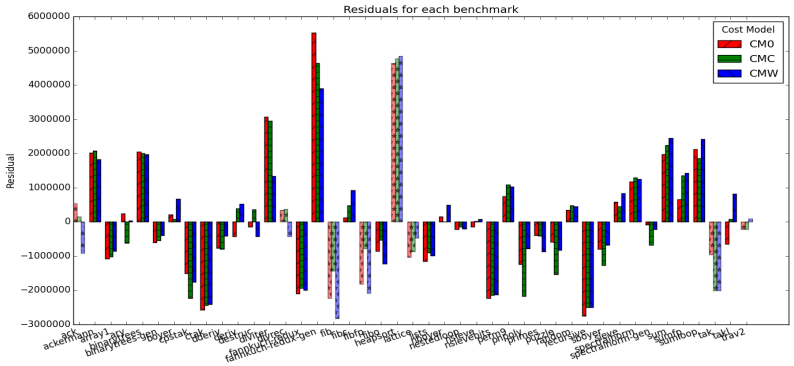

Fig. 7: Residuals for all benchmarks

Bibliography

[1] Ajitpar project. http://www.dcs.gla.ac.uk/ pmaier/AJITPar/ (2014), accessed: 2014-11-13

[2] (2015), http://benchmarksgame.alioth.debian.org/

[3] (2015), https://github.com/magnusmorton/pycket

[4] pycket-bench. https://github.com/krono/pycket-bench (2015), accessed: 2015-03-31

[5] Aycock, J.: A brief history of just-in-time. ACM Computing Surveys (CSUR) 35(2), 97–113 (2003)

[6] Bauman, S., Bolz, C.F., Hirschfeld, R., Krilichev, V., Pape, T., Siek, J.G., Tobin-Hochstadt, S.: Pycket: A tracing JIT for a functional language. In: ICFP '15 (2015), to appear

[7] Bolz, C.F., Cuni, A., Fijalkowski, M., Rigo, A.: Tracing the meta-level: Pypy's tracing jit compiler. In: Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems. pp. 18–25. ACM (2009)

[8] Bolz, C.F., Kuhn, A., Lienhard, A., Matsakis, N.D., Nierstrasz, O., Renggli, L., Rigo, A., Verwaest, T.: Back to the future in one weekâĂŤimplementing a smalltalk vm in pypy. In: Self-Sustaining Systems, pp. 123–139. Springer (2008)

[9] Bolz, C.F., Pape, T., Siek, J., Tobin-Hochstadt, S.: Meta-tracing makes a fast racket (2014)

[10] Bozó, I., Fordós, V., Horvath, Z., Tóth, M., Horpácsi, D., Kozsik, T., Köszegi, J., Barwell, A., Brown, C., Hammond, K.: Discovering parallel pattern candidates in erlang. In: Proceedings of the Thirteenth ACM SIGPLAN workshop on Erlang. pp. 13–23. ACM (2014)

[11] Brady, E., Hammond, K.: A dependently typed framework for static analysis of program execution costs. In: Implementation and Application of Functional Languages, pp. 74–90. Springer (2006)

[12] Brandolese, C., Fornaciari, W., Salice, F., Sciuto, D.: Source-level execution time estimation of c programs. In: Proceedings of the ninth international symposium on Hardware/software codesign. pp. 98–103. ACM (2001)

[13] Brown, C., Danelutto, M., Hammond, K., Kilpatrick, P., Elliott, A.: Cost-directed refactoring for parallel erlang programs. International Journal of Parallel Programming 42(4), 564–582 (2014)

[14] Cole, M.I.: Algorithmic skeletons: structured management of parallel computation. Pitman London (1989)

[15] Crawley, M.J.: Statistics: an introduction using R. John Wiley & Sons (2014)

[16] Dal Lago, U., Peña, R.: Foundational and Practical Aspects of Resource Analysis: Third International Workshop, FOPARA 2013, Bertinoro, Italy, August 29-31, 2013, Revised Selected Papers, vol. 8552. Springer (2014)

[17] Ferdinand, C., Heckmann, R.: ait: Worst-case execution time prediction by static program analysis. In: Building the Information Society, pp. 377–383. Springer (2004)
[18] Foundation, P.S.: Python (March 2015), http://python.org
[19] Group, K.O.W., et al.: The opencl specification. version 1(29), 8 (2008)
[20] Jones, S.L.P.: Compiling haskell by program transformation: A report from the trenches. In: Programming Languages and Systems - ESOP'96, pp. 18–44. Springer (1996)
[21] Jones, S.P., Tolmach, A., Hoare, T.: Playing by the rules: rewriting as a practical optimisation technique in ghc. In: Haskell workshop. vol. 1, pp. 203–233 (2001)
[22] JSON: Json (March 2015), http://www.json.org
[23] McCabe, T.J.: A complexity measure. Software Engineering, IEEE Transactions on (4), 308–320 (1976)
[24] Project, T.: Topaz (March 2015), https://github.com/topazproject/topaz
[25] Russell, S., Norvig, P.: Artificial intelligence: a modern approach (1995)
[26] Scaife, N., Horiguchi, S., Michaelson, G., Bristow, P.: A parallel sml compiler based on algorithmic skeletons. Journal of Functional Programming 15(04), 615–650 (2005)
[27] Steele Jr, G.L., Sussman, G.J.: The revised report on scheme: A dialect of lisp. Tech. rep., DTIC Document (1978)
[28] team, P.: Pypy (March 2015), http://pypy.org
[29] Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., et al.: The worst-case execution-time problem - overview of methods and survey of tools. ACM Transactions on Embedded Computing Systems (TECS) 7(3), 36 (2008)